Evolutionary Program Sketching

Iwo Błądek, Krzysztof Krawiec

Poznan University of Technology

19.04, Evostar 2017

Outline of the Presentation

1. Introduction
2. Satisfiability Modulo Theories (SMT)
3. SMT-Based Synthesis
4. Evolutionary Program Sketching

Software Engineering point of view

In SE programs are expected to be:

1. correct (no bugs).
2. easy to understand for the programmer.
3. as efficient as possible without breaking the above constraints. :)

Software Engineering point of view

In SE programs are expected to be:

1. correct (no bugs).
2. easy to understand for the programmer.
3. as efficient as possible without breaking the above constraints. :)

Q: How close at the moment is GP to meeting those objectives in practice?

Software Engineering point of view

In SE programs are expected to be:

1. correct (no bugs).
2. easy to understand for the programmer.
3. as efficient as possible without breaking the above constraints. :)

Q: How close at the moment is GP to meeting those objectives in practice?

A: Not very close.

1. Correctness outside of test cases not specified (induction).
2. Results hard to understand.
3. Resulting programs may be efficient (provided this is mandated by fitness function).

Arbitrary constants

Problem definition

- Synthesizing programs containing constants is problematic. For example, the target optimal program may be:

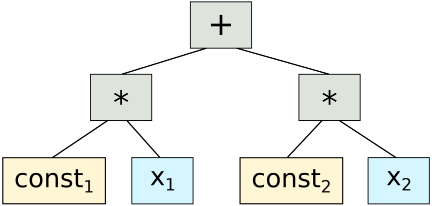
$$f(x_1, x_2, x_3) = 100017x_1 + 128.2x_2 - 0.12782x_3 + 190$$

- **Our hypothesis:** Constants may be found by the dedicated solver, thus increasing efficiency of GP, and potentially making programs easier to understand (constants may be derived directly).

Sketch

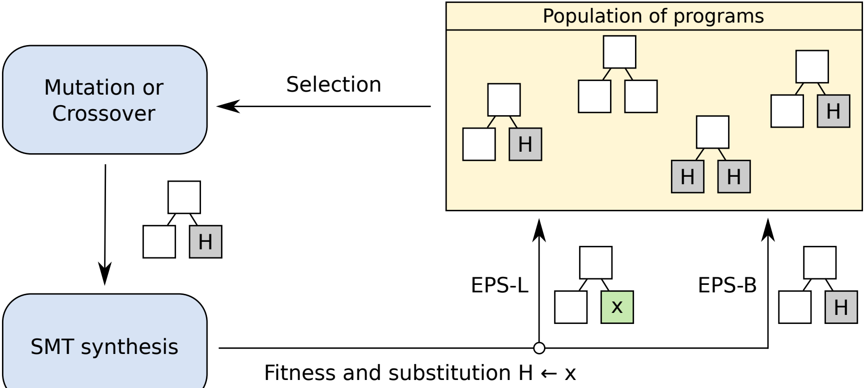
Sketch* – a partial program, in which certain parts are unspecified. Content of those parts will be found by an **SMT solver**. In general, holes may stand for any subprogram.

Example:



* (Solar-Lezama et al., 2006, *Combinatorial Sketching for Finite Programs*)

Evolutionary Program Sketching



Outline of the Presentation

1. Introduction
2. Satisfiability Modulo Theories (SMT)
3. SMT-Based Synthesis
4. Evolutionary Program Sketching

Satisfiability Problem (SAT)

Question: Is the given logical formula satisfiable?

Examples:

$\neg a \;\vee\; b$
SAT: $a = \mathit{false},\; b = \mathit{true}$

$a \;\wedge\; \neg a \;\wedge\; b$
UNSAT

Satisfiability Modulo Theories (SMT)

Question: Is the given logical formula satisfiable under the *theory* T, which defines semantics of a certain set of functions?

Examples:

`QF_LIA` (Quantifier-Free Linear Integer Arithmetic)
$x, y, z \in \mathbb{Z}$
$a \in \{false, true\}$

$(10 \cdot x = 20) \ \wedge \ a$
SAT: $x = 2, \ a = true$

$(x < y) \ \wedge \ (y < z) \ \wedge \ (z < x)$
UNSAT

$(x \leq y) \ \wedge \ (y \leq z) \ \wedge \ (z \leq x)$
SAT: $x = 0, \ y = 0, \ z = 0$

Satisfiability Modulo Theories (SMT)

Question: Is the given logical formula satisfiable under the *theory* T, which defines semantics of a certain set of functions?

Examples:

`NIA` (Non-Linear Integer Arithmetic)
$x, y \in \mathbb{Z}$

$x^2 + 1 \leq 2 \cdot x$
SAT: $x = 1$

$\forall_{x,y}\ (x + y)^2 > x^2 + y^2$
UNSAT

SMT Solvers

SMT Solver – any software that can check satisfiability of formulas modulo the given theory.

Notable SMT solvers:

- CVC4 (open source)
- MathSAT (free for non-commercial use)
- Z3 (open source, project of Microsoft Research)

SMT-LIB language – language created to standardize interaction with different SMT solvers.

Outline of the Presentation

1. Introduction
2. Satisfiability Modulo Theories (SMT)
3. SMT-Based Synthesis
4. Evolutionary Program Sketching

Notation

- **pre**(in) – **precondition**
  Behavior of the program is specified only for inputs that satisfy this formula.

  $$e.g. \quad in_1 \geq 0 \ \wedge \ in_2 \geq 0$$

- **program**(in, out) – **encoding of the program**
  Ensures that *out* must have the same values as it would have if the original program was executed.

  $$e.g. \quad out_1 = in_1 + in_1 - (in_2 - in_2)$$

- **post**(in, out) – **postcondition**
  Describes the expected behavior of the program.

  $$e.g. \quad out_1 \geq in_1 + in_2 \quad \wedge \quad out_1 \leq 2 \cdot (in_1 + in_2)$$

Program Synthesis Formula

Program synthesis formula:

$$\exists_{svars}\forall_{in,out}\quad \textbf{pre}(in) \land \textbf{program}(svars, in, out) \implies \textbf{post}(in, out)$$

where:

- *svars* – **Structural variables**
  Variables controling the shape of the synthesized program.

An Example of SMT Synthesis

Task: Compute a maximum of two numbers x and y.

General structure of the solution (sketch):

```
if (H1):
  res = H2
else:
  res = H3
```

H1, H2, H3 – holes to be filled by the synthesizer.

An Example of SMT Synthesis

Program’s encoding in the SMT-LIB language:

```
(assert
(forall ((x Int)(y Int)(res Int)(|res’’| Int)(|res’| Int))
(=>
  ;PROGRAM:
    (and
      (=> (H1Start0 x y)  ;TRUE IF BRANCH
          (and (= res (H2Start0 x y)) (= |res’’| res)))
      (=> (not (H1Start0 x y))  ;ELSE IF BRANCH
          (and (= |res’| (H3Start0 x y)) (= |res’’| |res’|)))
    )
  ;POSTCONDITION:
    (and (>= |res’’| x) (>= |res’’| y)
        (or (= |res’’| x) (= |res’’| y)))
    )
))
```

An Example of SMT Synthesis

Encoding of hole's grammar (for H2):

```
(define-fun H2Start0 ((x Int)(y Int)) Int
  (ite (= H2Start0_r0 0)
    H2Start0_Int0
    (ite (= H2Start0_r0 1)
      x
      (ite (= H2Start0_r0 2)
        y
        (ite (= H2Start0_r0 3)
          (+ x y)
          (ite (= H2Start0_r0 4)
            (- x y)
            ... )
```

Structural variables:

H2Start0_r0, H2Start0_Int0

An Example of SMT Synthesis

Model returned by SMT solver:

```
(model
  (define-fun H1Start0_Bool0 () Bool false)
  (define-fun H1Start0_r0 () Int 2)
  (define-fun H1Start0_Int0 () Int (- 2))
  (define-fun H2Start0_Int0 () Int 2)
  (define-fun H2Start0_r0 () Int 1)
  ...
)
```

Final synthesized code, created from model:

```
if (>= x y):
  res = x
else:
  res = y
```

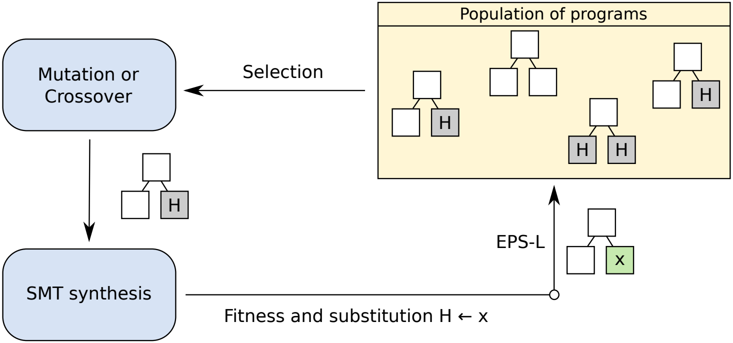
Outline of the Presentation

1. Introduction
2. Satisfiability Modulo Theories (SMT)
3. SMT-Based Synthesis
4. Evolutionary Program Sketching

EPS

EPS-L – “Lamarckian” EPS

After evaluation holes are permanently filled with content found by the solver. New holes may be introduced only via mutations.



Population of programs

Selection

Mutation or Crossover

SMT synthesis

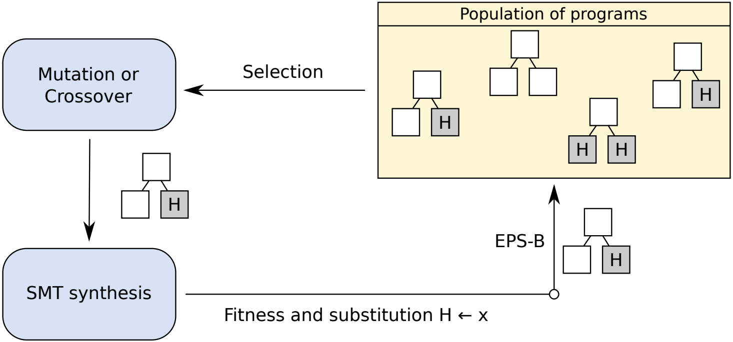
Fitness and substitution H ← x

EPS-L

EPS

EPS-B – "Baldwinian" EPS

After evaluation holes remain in a program. Content found by the solver is discarded.



Population of programs

Mutation or Crossover

Selection

SMT synthesis

EPS-B

Fitness and substitution H ← x

EPS

Types of holes

C – Constant holes
Can be filled with an arbitrary integer constant.

V – Variable holes
Can be filled with one of the input variables.

CV – Constant & Variable holes
Can be filled with either an integer constant or an input variable.

Experiments

Benchmarks

Benchmark	*#vars*	*Formula*	*#tests*
`Keijzer12`	2	$x_1^4 - x_1^3 + x_2^2/2 - x_2$	49
`Koza1`	1	$x^4 + x^3 + x^2 + x$	11
`Koza1-p`		$3x^4 - 2x^3 + 6x^2 + 3x - 4$	
`Koza1-2D`	2	$x_1^4 + x_2^3 + x_1^2 + x_2$	49
`Koza1-p-2D`		$3x_1^4 - 2x_2^3 + 6x_1^2 + 3x_2 - 4$	

Logic: `NIA` (Non-linear Integer Arithmetic)

Experiments

Evolution parameters

Parameter	*Value*
Number of runs	100
Maximum number of generations	100
Population size	250
Maximum height of initial programs	4
Maximum height of subprograms inserted by mutation	4
Constant terminals drawn from interval	$[0,5]$
Probability of mutation	0.5
Probability of crossover	0.5
Tournament size	7
Solver timeout [ms]	1500

Experiments

Number of optimal solutions found (/100)

	GP			EPS-L			EPS-B		
	GP	GP_T	GP_{5000}	*c*	*v*	*cv*	*c*	*v*	*cv*
`Keijzer12`	0	0	5	0	0	1	39	1	0
`Koza1`	19	68	96	33	-	32	100	-	100
`Koza1-p`	0	0	0	5	-	3	100	-	100
`Koza1-2D`	1	12	20	2	0	11	80	21	23
`Koza1-p-2D`	0	0	0	0	0	1	75	0	0

Average runtime [s]

	GP			EPS-L			EPS-B		
	GP	GP_T	GP_{5000}	*c*	*v*	*cv*	*c*	*v*	*cv*
`Keijzer12`	15	11331	493	772	488	1579	15440	21173	28354
`Koza1`	5	291	46	700	-	801	652	-	696
`Koza1-p`	5	963	344	892	-	972	978	-	982
`Koza1-2D`	16	7636	432	793	479	1791	9077	16281	23034
`Koza1-p-2D`	15	9206	515	750	511	1726	11986	12391	27875

Experiments

Ratio of UNKNOWN solver response

	EPS-L			EPS-B		
	c	*v*	*cv*	*c*	*v*	*cv*
`Keijzer12`	0.058	0.004	0.104	0.229	0.106	0.297
`Koza1`	0.080	-	0.058	0.127	-	0.120
`Koza1-p`	0.078	-	0.060	0.113	-	0.112
`Koza1-2D`	0.065	0.006	0.118	0.276	0.117	0.372
`Koza1-p-2D`	0.062	0.004	0.112	0.301	0.051	0.407

Source code:

`https://github.com/iwob/EPS`

Summary

EPS:

1. Evolution responsible for program *structure*, SMT solver cares about the details (fills in the gaps).
2. Improves over standard GP.
3. Works particularly well for constants.

Part of our agenda of combining heuristics with SMT solvers for program synthesis.

Final words

Thank you for your attention!

Our next paper:
`GECCO 2017`, **“Counterexample-Driven Genetic Programming”**
(Krawiec, Błądek, Swan)